**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE**
**BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES**

| | | | |
|---|---|---|---|
| Application. No: 09/941,057<br>Filed: August 28, 2001<br>Inventor(s):<br>Steven W. Rogers and Jeffrey L. Kodosky<br><br>Title: SYSTEM AND METHOD FOR ANALYZING A GRAPHICAL PROGRAM USING DEBUGGING GRAPHICAL PROGRAMS | §<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§<br>§ | Examiner:<br>Group/Art Unit:<br>Atty. Dkt. No: | Kendall, Chuck O.<br>2192<br>5150-59901 |

## APPEAL BRIEF

Dear Sir/Madam:

This Appeal Brief is submitted in response to the Notification of Non-Compliant Appeal Brief mailed by the PTO on 03/01/2007.

Further to the Notice of Appeal filed October 25, 2006 and in response to the Notice of Panel Decision from Pre-Appeal Brief Review of December 7, 2006, Appellant presents this Appeal Brief. Appellant respectfully requests that this appeal be considered by the Board of Patent Appeals and Interferences.

## I. REAL PARTY IN INTEREST

The subject application is owned by National Instruments Corporation, a corporation organized and existing under and by virtue of the laws of the State of Delaware, and having its principal place of business at 11500 N. MoPac Expressway, Bldg. B, Austin, Texas 78759-3504.

## II. RELATED APPEALS AND INTERFERENCES

No related appeals or interferences are known which would directly affect or be directly affected by or have a bearing on the Board's decision in this appeal.

## III. STATUS OF CLAIMS

Claim 12 was cancelled. Claims 1-11 and 13-40 are pending in the case and are the subject of this appeal. Claims 1-7, 9-11, 13-15, and 17-40 stand rejected under 35 U.S.C. 102 (e) and claims 8 and 16 stand rejected under 35 U.S.C. 103(a). Pending claims 1-11 and 13-40 are included in the attached Claims Appendix.

## IV. STATUS OF AMENDMENTS

No amendments to the claims have been filed subsequent to the rejection in the Office Action of July 27, 2006. The Claims Appendix reflects the current state of the claims.

## V. SUMMARY OF THE CLAIMED SUBJECT MATTER

The present invention relates to the field of graphical programming, and more particularly, to a system and method for enabling a graphical program to use and implement debugging graphical programs.

Independent claim 1 recites a method for configuring a debugging node for a graphical program, where the method executes on a first computer system.

A first graphical program is created using a graphical programming development environment, where creating includes interconnecting at least two of a first plurality of graphical program nodes or icons, where the first graphical program includes the first plurality of interconnected graphical program nodes or icons which graphically represents functionality of the first graphical program, and where the first graphical program is executable by a computer system to perform the functionality. For example, the user can create a first graphical program by connecting two or more graphical program nodes, where the first graphical program is executable by the computer. *See, e.g., p. 23:10-16, p. 24:11-23, p. 31:26 – p. 32:13; step 304 of Figure 4, Figure 9.* The first graphical program is stored in a memory. For example, the first graphical program can be stored in memory of the first computer. *See, e.g., p. 23:10-16, p. 24:11-23; step 304 of Figure 4.*

A debugging graphical program is created using the graphical programming development environment. *See, e.g., p. 23:10 – p. 24:10, p. 31:26 – p. 32:13, step 302 of Figure 4, Figure 9.* This debugging graphical program is associated at a debugging location in the first graphical program without modifying the functionality of the first graphical program. *See, e.g., p. 25:3 – p. 27:5, p. 30:28 – p. 32:13, step 308 of Figure 4.* The debugging graphical program includes a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the debugging graphical program. *See, e.g., p. 31:3-18, Figure 7.* The debugging graphical program is executable during execution of the first graphical program to aid in debugging at least a portion of the first graphical program. *See, e.g., p. 27:9-17, p. 31:26 – p. 32:13, step 312 of Figure 4, Figure 9.*

Independent claim 23 recites a method for configuring a debugging node for a graphical program, where the method executes on a first computer system.

A first graphical program is executed up to a debugging location, where the first graphical program includes a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and where the first graphical program is executable by a computer system to perform the functionality. *See, e.g., p. 23:10-16, p. 24:16-23.* The first graphical program generates data at the debugging location, where the first graphical program was created using a graphical programming development environment. For example, the user can create a first graphical program by connecting two or more graphical program nodes, where the first graphical program is executable by the computer. *See, e.g., p. 31:26 – p. 32:13; step 304 of Figure 4, Figure 9.*

The data is provided to a debugging graphical program, where the debugging graphical program includes a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the debugging graphical program, where the debugging graphical program was created using the graphical programming development environment. *See, e.g., p. 27:20 – p. 28:5; step 400 of Figure 5.* The debugging graphical program is executed, where the debugging graphical program uses the data. *See, e.g., p. 28:6-20; step 402 of Figure 5.* The debugging graphical program generates debugging results. *See, e.g., p. 28:21 – p. 29:10; step 404 of Figure 5.* The use of the debugging graphical program does not require modification or re-compilation of the first graphical program. *See, e.g., p. 25:3 – p. 27:5; step 308 of Figure 4.*

Independent claim 27 is directed to a method for configuring a graphical program with a debugging graphical program similar to that of claim 1.

A first graphical program is stored in a memory of a computer system, where the first graphical program comprises a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and where the first graphical program was created using a graphical programming development environment. *See, e.g., p. 23:10-16, p. 24:11-23, p. 31:26 – p. 32:13; step 304 of Figure 4, Figure 9.* The first graphical program is executable by the computer system to perform the functionality. *See, e.g., p. 27:6-8, Step 310 of Figure 27.*

A second graphical program is created using the graphical programming development environment, and the second graphical program includes a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program. *See, e.g., p. 23:10 – p. 24:10, p. 31:3-18, step 302 of Figure 4, Figure 7.* The second graphical program is associated at a location in the first graphical program, where associating does not modify the functionality of the first graphical program. *See, e.g., p. 25:3 – p. 27:6, p. 31:26 – p. 32:13, step 308 of Figure 4, Figure 9.*

The second graphical program is executable during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program. *See, e.g., p. 28:16 – p. 30:20; Figure 5.*

Independent claim 31 is directed to a memory medium for configuring a graphical program with a debugging graphical program similar to that of claim 27.

The memory medium includes program instructions for analyzing a first graphical program, wherein the program instructions are executable to implement storing a first graphical program in a memory of a computer system, where the first graphical program comprises a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and the first graphical program was created using a graphical programming development environment. *See, e.g., p. 23:10-16, p. 24:11-23, p. 31:26 – p. 32:13; step 304 of Figure 4, Figure 9.* The first graphical program is executable by the computer system to perform the functionality. *See, e.g., p. 27:6-8, Step 310 of Figure 27.*

The memory medium includes program instructions for associating the second graphical program at a location in the first graphical program, where associating does not modify the functionality of the first graphical program. *See, e.g., p. 25:3 – p. 27:6, p. 31:26 – p. 32:13, step 308 of Figure 4, Figure 9.* The second graphical program is created using the graphical programming development environment, and the second graphical program includes a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program. *See, e.g., p. 23:10 – p. 24:10, p. 31:3-18, step 302 of Figure 4, Figure 7.*

The memory medium includes program instructions for executing the second graphical program during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program. *See, e.g., p. 28:16 – p. 30:20; Figure 5.*

Independent claim 35 is directed to a memory medium for configuring a graphical program with a debugging graphical program.

The memory medium includes a first graphical program, where the first graphical program comprises a first plurality of interconnected nodes which visually indicate functionality of the first graphical program, where the first graphical program was created using a graphical programming development environment. *See, e.g., p. 23:10-16, p. 24:11-23, p. 31:26 – p. 32:13; step 304 of Figure 4, Figure 9.* The first graphical program is executable by the computer system to perform the functionality. *See, e.g., p. 27:6-8, Step 310 of Figure 27.*

The memory medium includes a second graphical program, where the second graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program, where the second graphical program was created using the graphical programming development environment. *See, e.g., p. 23:10 – p. 24:10, p. 31:3-18, step 302 of Figure 4, Figure 7.*

The memory medium includes a data structure which is operable to store information associating the second graphical program with a location in the first graphical program, where the functionality of the first graphical program is not modified by the second graphical program. *See, e.g., p. 25:3 – p. 27:14, p. 31:26 – p. 32:13, step 308 of Figure 4, Figure 9.* The second graphical program is executable during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program. *See, e.g., p. 28:16 – p. 30:20; Figure 5.*

## VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

Claims 1-7, 9-15, and 17-40 stand rejected under 35 U.S.C. 102 (e) as being unpatentable over Leask et al. (U.S. Patent No. 6,412,106, "Leask").

Claim 8 stands rejected under 35 U.S.C. 103(a) as being unpatentable over Leask in view of McKee et al. (U.S. Patent No. 5,915,114, "McKee").

Claim 16 stands rejected under 35 U.S.C. 103(a) as being unpatentable over Leask in view of Kodosky (U.S. Patent Application No. 2003/0037322, "Kodosky").

## VII. ARGUMENT

**First Ground of Rejection**

Claims 1-7, 9-11, 13-15, and 17-40 stand rejected under 35 U.S.C. 102 (e) as being unpatentable over Leask. Appellant respectfully traverses this rejection for the following reasons. Different groups of claims are addressed under their respective subheadings.

**Claims 1, 2, 13, 17, 18, 19, 20, 21, 22, 27, 28, 31, 32, 37, 38, 39, and 40**

Claim 1 is separately patentable because the cited reference does not teach or suggest the limitations recited in this claim. Leask teaches using debugging tools for debugging a graphical representation of a program, where the debugging tools "may be represented graphically to indicate which tools are currently set in the program" (Leask col. 7, lines 26-29).

In asserting that Leask teaches the elements of claim 1, the Office Action cites col. 11:45-50, col. 7:7-20 & 10:1-5, elements 410, 412, 414, 416, and 418 of Figure 5, and col. 13:23-30, but fails to explain how any of the cited portions teach a debugging graphical program or creating a debugging graphical program using a graphical programming environment. The description for the cited elements 410, 412, 414, 416, and 418 of Figure 5, described in Leask col. 12, lines 18-36, reads:

> Turning to FIG. 5, an exemplary graphical debugging environment is shown. Call flow diagram 300, as discussed above with reference to FIG. 4, is shown. The graphical debugging environment is capable of displaying a graphical representation of an application program much like a graphical development environment. Additionally, various testing and debugging tools 402-418 that may be offered to a developer in such a graphical debugging environment are illustrated. The function of each testing and debugging tool will be discussed in greater detail hereafter. It is important to realize at this point that debugging tools may be utilized within the graphical debugging environment without requiring a programmer to interact with the underlying textual source code of an application program. That is, debugging tools may be utilized by a programmer to debug an application at the graphical representation level, rather than at the lower textual source code level. Accordingly, the graphical debugging environment allows a programmer to utilize graphical debugging tools to perform highlevel debugging.

Appellant notes that the cited elements 410, 412, 414, 416, and 418 of Leask are icons that represent executable functionality, and are not graphical programs that can be created by the user as recited in claim 1. Furthermore, the icons for the debugging tools of Leask do not contain interconnected graphical program nodes as required by claim 1 to be a graphical program. In other words, the debugging tools of Leask are not created using the same graphical programming environment as the graphical programs of Figures 3 or 4 of Leask. Figure 5 of Leask shows the debugging environment that debugs the graphical program of Figure 4 using debugging tools.

Claim 1 recites that the debugging graphical program is created using the graphical programming environment. Leask does not teach or suggest that any of the graphical representations, i.e., icons, of the debugging tools (i.e., the cited elements 410, 412, 414, 416, or 418 of Figure 5) were created using a graphical programming development environment. Specifically, the cited passage (col. 11, lines 45-50) does not show that the debugging icons 410, 412, 414, 416, or 418 of Leask were created using a graphical programming environment. Specifically, Leask at column 11, lines 45-53 reads:

> Icon 330 represents the end of the call. Thus, call flow diagram 300 is an example of a graphical development environment in which a computer application is represented graphically. It should be understood that an application program need not be represented exactly as shown in FIGS. 3 and 4. Rather, an application may be represented in any manner, and the graphical debugger disclosed herein may be implemented to debug any manner of graphical representation.

The cited passage only teaches an exemplary graphical program of Leask, and it does not teach that the debugging tools of Leask were created using the graphical programming environment. Appellant notes that there are two graphical programs recited in claim 1, the first graphical program and the debugging graphical program. The cited passage only teaches that the debugging tools 410, 412, 414, 416, or 418 can debug a graphical program, such as the one created using the graphical development environment shown in Figure 5 of Leask. In other words, although the debugging tools of Leask are represented as graphical icons (e.g., disclosed in col. 7, line 20-35), the icons are not themselves graphical programs. Leask simply does not teach or suggest that a user can create or modify a graphical program for a debugging icon. Thus, neither the icons that

represent the debugging tools nor the debugging tools of Leask are executable graphical programs.

Appellant additionally notes that Figures 8 and 9 of Leask also are not graphical programs that contain graphical program elements as recited in claim 1, but instead are only diagrams that show the operation of the debugging engine of Leask.

Therefore Leask does not teach creating a debugging graphical program using a plurality of interconnected graphical program nodes or icons that graphically represent the functionality of the debugging graphical program. Thus, the icon, i.e., a graphical representation of the debugging tool of Leask, is not equivalent to the debugging graphical program of claim 1.

Thus, Appellant respectfully submits that Leask fails to disclose all the features and limitations of claim 1, and so, for at least the reasons provided above, Appellant submits that claim 1 is patentably distinct and non-obvious over Leask, and thus allowable.

**<u>Claim 3, 29, and 33</u>**

In addition to the novel limitations of claim 1, Leask fails to teach or suggest "executing the first graphical program up to the debugging location; executing the debugging graphical program after executing the first graphical program up to the debugging location; and the debugging graphical program generating debugging results, wherein the debugging results are useful in analyzing at least a portion of the first graphical program" as recited in claim 3.

In asserting that Leask teaches this feature, the Office Action cites col. 10:1-5, and col. 12:50-57, which read:

> An application program to be debugged may be running either on the local system $100_{LOCAL}$ or on a remote system, such as $100_{REMOTE}$ shown in FIG. 1. Accordingly, the graphical debugger program running on local computer $100_{LOCAL}$ may be utilized to debug locally stored programs or programs stored at remote locations (e.g., $100_{REMOTE}$) via network 108.

(col. 10:1-8), and

> One debugging tool that may be available in the graphical debugging environment is a breakpoint. A breakpoint may suspend an application program (i.e., make the application program stop advancing) or perform some other debug function whenever a certain point in the application

> program is reached, After the application is suspended a developer may analyze the states of various components of the application program, such as the value of variables used in the program. A developer may even have the capability to modify values stored in variables, while the application program is suspended. Breakpoints are one of the primary debug tools commonly utilized by developers in debugging application programs.

(col. 12:47-57)

Appellant notes that the first cited passage (col. 10:1-8) discloses distributed operation of the debugger of Leask. The first cited passage does not teach or suggest "executing the first graphical program up to the debugging location" nor does it teach or suggest "executing the debugging graphical program after executing the first graphical program up to the debugging location." There is no mention in the first cited passage of executing the first graphical program up to a debugging location. There is also no mention in the first cited passage of executing the debugging graphical program. Furthermore, as noted above with reference to claim 1, Leask does not teach that the debugging icons are graphical programs.

Appellant notes that the second cited passage discloses the operation of a breakpoint. Breakpoints are well known in the art of programming. However, the second cited passage does not teach the novel elements of claim 3, which are that the debugging graphical program generates debugging results which are useful in analyzing a portion of the first graphical program. Instead, the second cited passage teaches that when execution of the graphical program reaches the breakpoint, the user of the Leask debugging system can manually examine variables.

Claim 3 does not claim that the debugging graphical program is a breakpoint. Instead, claim 3 recites that the execution of the debugging graphical program produces results that are useful in analyzing the first graphical program. For example, Figures 6 and 8 of the Application show exemplary debugging results produced by an exemplary debugging graphical program. The exemplary debugging results show an error condition with a code of 43. The debugging results are obtained by the user supplied functionality in the debugging graphical program. This is very different from the developer examining variables while the graphical program is halted using a breakpoint, as described in the second cited passage of Leask.

Nowhere does Leask teach or suggest executing the first graphical program up to the debugging location, executing the debugging graphical program after executing the first graphical program up to the debugging location, and the debugging graphical program generating debugging results, where the debugging results are useful in analyzing at least a portion of the first graphical program. Thus, Leask fails to teach this feature of claim 3, and so claim 3 is allowable.

**<u>Claim 4</u>**

In addition to the novel limitations of claims 1 and 3, Leask fails to teach or suggest "completing execution of the first graphical program based on the debugging results of said executing the debugging graphical program", as recited in claim 4.

In asserting that Leask teaches this feature, the Office Action cites col. 7:30-35, which reads:

> In a preferred embodiment, the debug tools, such as breakpoints, may be set while the application program being debugged is executing, without requiring that the application be halted. Likewise, a preferred embodiment also allows for such debugging tools to be modified or removed from the application program without requiring that the application be halted.

Appellant respectfully submits that the cited passage only discloses that a breakpoint may be inserted into a graphical program being debugged without halting the graphical program, and not the novel elements of claim 4. Specifically, nowhere does Leask disclose that the execution of the first graphical program is completed based on the debugging results of the debugging graphical program, i.e., Leask does not teach that execution of the graphical program is completed based on the results of the execution of the debugging icon of Leask.

Thus, Leask fails to teach this feature of claim 4, and so claim 4 is allowable.

**<u>Claim 5</u>**

In addition to the novel limitations of claims 1 and 3, Leask fails to teach or suggest "wherein said executing the debugging graphical program includes displaying the debugging results of the debugging graphical program", as recited in claim 5.

In asserting that Leask teaches this feature, the Office Action cites col. 7:8-20, which reads:

> In a preferred embodiment, <u>a graphical representation of an application program to be debugged is displayed to a developer</u>. For example, underlying source code of the application program may be represented by interconnected icons. Such a graphical representation may represent a computer program that is stored either locally or remotely. Additionally, such a graphical representation may represent a program that is currently executing either locally or remotely, or the graphical representation may represent a program that is not currently executing. A developer may then utilize the graphical debugger environment to insert debugging tools, such as breakpoints, directly into the graphical representation of the application program.

Appellant respectfully submits that the cited passage describes displaying a graphical representation of the graphical program that is being debugged, and not debugging results of the debugging graphical program. Even if a distinction between a debugging graphical program of claim 1 and the debugging icon of Leask is disregarded, the cited passage of Leask still does not teach displaying debugging results of the debugging icon of Leask.

Nowhere does Leask teach or suggest executing the debugging graphical program that includes displaying the debugging results of the debugging graphical program. Thus, Leask fails to teach this feature of claim 5, and so claim 5 is allowable.

**<u>Claim 6</u>**

In addition to the novel limitations of claims 1 and 3, Leask fails to teach or suggest "wherein said executing the debugging graphical program comprises:

receiving data from the first graphical program; and

performing one or more of:

displaying the data from the first graphical program; and/or

logging the data from the first graphical program to a file," as recited in claim 6.

In asserting that Leask teaches this feature, the Office Action cites col. 19:30-40, col. 21:40-43, and col. 26:54, which read respectively:

> Once a Block ID has been selected, the user requests to have a debug tool inserted, removed, or modified for the selected Block ID. For example, a developer may select a particular icon and insert a breakpoint as shown in FIG. 5, and the developer's request for a breakpoint at that particular icon's block ID is received by the graphical debugger. Thereafter, <u>the graphical debugger communicates the requested service (i.e., inserting/removing/modifying a debug tool) to the Debug Engine</u> at block 708.

(col. 19:30-40).

> Accordingly, each debug tool requested by a developer may have an <u>indicator displayed within the graphical representation of the application program</u> and may be communicated to the debug engine.

(col. 21:39-43)

> "a memory for <u>storing computer executable program code</u>;"

(col. 26:54)

The first cited passage describes that the graphical debugger communicates a requested service to the debug engine of Leask, where the service can be inserting, removing, or modifying a debug tool. Appellant respectfully notes that the first cited passage does not teach or suggest the debugging graphical program receiving data from the first graphical program. Appellant respectfully notes that even if a distinction between a debugging graphical program of claim 1 and the debugging icon of Leask is disregarded, the cited passage of Leask still does not teach the debugging icon of Leask receiving data from the graphical program.

The second cited passage describes that a breakpoint may be indicated graphically in the graphical program of Leask. Specifically, the Summary of Leask describes that the graphical debugging environment "may display <u>indicators</u> illustrating where debug tools have been inserted within the application program." Therefore the cited indicator does not display data from the graphical program as a result of executing the debugging graphical program, but instead only indicates a location of the debugging icon in the graphical program of Leask.

The third cited passage is in fact an element of claim 44 of Leask. Appellant respectfully notes that the cited claim element describes using computer memory for storing executable program code, as described in the next element of claim 44 "said program code comprises code for representing said computer application program graphically on said display." Appellant notes that storing program code in memory is

very different from logging the data from the first graphical program to a file while executing the debugging graphical program. Appellant notes that the third cited passage does not teach or suggest logging data received from the graphical program.

Thus, Leask fails to teach this feature of claim 6, and so claim 6 is allowable.

**Claim 7**

In addition to the novel limitations of claims 1 and 3, Leask fails to teach or suggest that executing the debugging graphical program includes "receiving data from the first graphical program, generating statistics based on the received data, and displaying the statistics," as recited in claim 7.

In asserting that Leask teaches this feature, the Office Action cites col. 10:5-10, col. 3:15-23, and col. 7:7-10, which read respectively:

> An application program to be debugged may be running either on the local system $100_{LOCAL}$ or on a remote system, such as $100_{REMOTE}$ shown in FIG. 1. Accordingly, the graphical debugger program running on local computer $100_{LOCAL}$ may be utilized to debug locally stored programs or programs stored at remote locations (e.g., $100_{REMOTE}$) via network 108.
> Turning to FIG. 2, an example of how a prior art textual debugger may be utilized is shown.

(col. 10:1-10)

> In most systems, there is a close mapping of program image onto what the user perceives as a user process. The object file also contains a table that maps some of the original source information, as variable and function names, onto addresses, offsets, sizes, and other pertinent properties of the program image. This so-called symbol table is usually not made part of the program image itself, but remains in the object file where other programs (like the debugger) can read and analyze it.

(col. 3:15-23)

> In a preferred embodiment, a graphical representation of an application program to be debugged is displayed to a developer.

(col. 7:7-10)

Appellant notes that the first cited passage (col. 10:5-10) discloses distributed operation of the debugger of Leask. The first cited passage does not teach or suggest "receiving data from the first graphical program."

Appellant notes that the second cited passage (col. 3:15-23) discloses generating a symbol table or a mapping of a program image into a user process. The second cited passage does not teach or suggest generating statistics based on the received data.

Appellant notes that the third cited passage (col. 7:7-10) discloses displaying the graphical program to be debugged. Appellant notes that the third cited passage does not teach or suggest displaying any data, results, or statistics from the debugging graphical program.

Thus, Leask fails to teach this feature of claim 7, and so claim 7 is allowable.

**Claim 9**

In addition to the novel limitations of claims 1 and 3, Leask fails to teach or suggest that a complete execution of the first graphical program is "performed in a single stepping mode based on the debugging results of said executing the debugging graphical program", as recited in claim 9.

In asserting that Leask teaches this feature, the Office Action cites elements 414 and 416 of figure 5 and their respective description (col. 16:17-49), which read:

> Step 414 is provided within the graphical debug environment. Step 414 may allow a developer to step through execution of the application program. For example, once a breakpoint fires to suspend execution of the application program, a developer may use Step 414 to step through the application program's execution. Accordingly, the application program may advance one icon (or one "block" of source code) each time that the developer activates Step 414. As a developer steps through each icon of a program the icon currently being executed may be highlighted or indicated in some other fashion to allow a developer to monitor the progress of the program's execution.
>
> Step Over 416 is also shown in FIG. 5. Step Over 416 may allow a developer to step from an icon (or "block" of source code) displayed on one level of the call flow to the next icon (or "block" of source code) displayed on the same level by stepping over all of the icons (or "blocks" of source code) on underlying levels of the call flow. For example, suppose a breakpoint is set at a graphical icon that contains several underlying icons. Once the breakpoint fires the application's execution will be suspended. A developer may use Step 414 to step through the execution of each graphical icon, including each underlying graphical icon. However, a developer may use Step Over 416 to advance the program's execution to the next graphical icon on the same level as the icon for which the breakpoint was set.

> Thus, Step Over 416 allows the application program to perform all of the underlying icons uninterrupted, and then suspend the application at the next icon at the same level as the icon for which the breakpoint was set. Again, the icon currently being executed may be highlighted or otherwise indicated to allow a developer to monitor the progress of the program's execution.

Appellant respectfully notes that elements 414 and 416 of Figure 5 illustrate debugger Step and Step Over functionality. Appellant respectfully notes that neither the Step nor the Step Over elements provide a single stepping mode that is performed based on the results of executing the debugging graphical program. In other words, although the Step or Step Over elements may allow the user to single-step a graphical program, Leask does not teach or suggest that entering the Single-Stepping mode is based on results of executing the debugging icon.

Thus, Leask fails to teach this feature of claim 9, and so claim 9 is allowable.

**Claims 10, 11, 30, 34, and 36**

In addition to the novel limitations of claim 1, Leask fails to teach or suggest "executing the first graphical program up to the debugging location, wherein the first graphical program generates data at the debugging location;

providing the data to the debugging graphical program;

executing the debugging graphical program, wherein the debugging graphical program uses the data;

the debugging graphical program generating debugging results;

based on the debugging results, performing one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 10.

In asserting that Leask teaches this feature, the Office Action cites col. 10:5-10, col. 3:15-23 & 10:1-5, col. 7:7-10, elements 914 and 916 of Figure 9, and elements 414 and 416 of Figure 5, which read:

> An application program to be debugged may be running either on the local system $100_{LOCAL}$ or on a remote system, such as $100_{REMOTE}$ shown in FIG. 1. Accordingly, the graphical debugger program running on local computer $100_{LOCAL}$ may be utilized to debug locally stored programs or programs stored at remote locations (e.g., $100_{REMOTE}$) via network 108.

Turning to FIG. 2, an example of how a prior art textual debugger may be utilized is shown.

(col. 10:1-10)

In most systems, there is a close mapping of program image onto what the user perceives as a user process. The object file also contains a table that maps some of the original source information, as variable and function names, onto addresses, offsets, sizes, and other pertinent properties of the program image. This so-called symbol table is usually not made part of the program image itself, but remains in the object file where other programs (like the debugger) can read and analyze it.

(col. 3:15-23)

In a preferred embodiment, a graphical representation of an application program to be debugged is displayed to a developer.

(col. 7:7-10)

Step 414 is provided within the graphical debug environment. Step 414 may allow a developer to step through execution of the application program. For example, once a breakpoint fires to suspend execution of the application program, a developer may use Step 414 to step through the application program's execution. Accordingly, the application program may advance one icon (or one "block" of source code) each time that the developer activates Step 414. As a developer steps through each icon of a program the icon currently being executed may be highlighted or indicated in some other fashion to allow a developer to monitor the progress of the program's execution.

Step Over 416 is also shown in FIG. 5. Step Over 416 may allow a developer to step from an icon (or "block" of source code) displayed on one level of the call flow to the next icon (or "block" of source code) displayed on the same level by stepping over all of the icons (or "blocks" of source code) on underlying levels of the call flow. For example, suppose a breakpoint is set at a graphical icon that contains several underlying icons. Once the breakpoint fires the application's execution will be suspended. A developer may use Step 414 to step through the execution of each graphical icon, including each underlying graphical icon. However, a developer may use Step Over 416 to advance the program's execution to the next graphical icon on the same level as the icon for which the breakpoint was set. Thus, Step Over 416 allows the application program to perform all of the underlying icons uninterrupted, and then suspend the application at the next icon at the same level as the icon for which the breakpoint was set. Again, the icon currently being executed may be highlighted or otherwise indicated to allow a developer to monitor the progress of the program's execution.

(col. 16:17-49)

If the debug engine determines at block 912 that the debug tool's condition is true, the debug engine performs the

> debug tool's function at block 914. For example, for a breakpoint the debug engine may halt execution at block 914.

(col. 21:20-25)

Appellant notes that the first cited passage (col. 10:5-10) discloses distributed operation of the debugger of Leask. The first cited passage does not teach or suggest "executing the first graphical program up to the debugging location, wherein the first graphical program generates data at the debugging location."

Appellant notes that the second cited passage (col. 3:15-23) teaches generating a symbol table and mapping of a program image into a user process by the use of an image file. The second cited passage does not teach or suggest "providing the data to the debugging graphical program." Appellant notes that the third cited passage (col. 10:1-5) discloses distributed operation of the debugger of Leask. The third cited passage does not teach or suggest "providing the data to the debugging graphical program." The second and third cited passages, taken singly or in combination, do not teach or suggest "providing the data to the debugging graphical program."

Appellant notes that fourth cited passage (col. 3:15-23) teaches generating a symbol table and mapping of a program image into a user process by the use of an image file. Appellant notes that the fifth cited passage (col. 7:7-10) teaches displaying the graphical program to be debugged on the screen. The fourth and fifth cited passages, taken singly or in combination, do not teach or suggest "executing the debugging graphical program, wherein the debugging graphical program uses the data." Instead, the fourth and fifth cited passages teach that an object file that contains the graphical program may also contain a symbol file that holds the variables for the graphical program. This graphical program can then be displayed on the screen in graphical form. Although these passages also describe that a debugger can access the symbol table, it is clear from this passage that the symbol table that contains the data is generated for the object file for the graphical program prior to its execution and it is not generated for the debugging icon.

Appellant notes that the sixth cited passage (col. 7:7-10) teaches displaying the graphical program to be debugged on the screen. The sixth cited passage does not teach the debugging graphical program generating debugging results. In fact, the sixth cited

passage does not mention the debugging icon at all, but instead discloses that the graphical to be debugged (i.e., by using the debugging icon), will simply be displayed on the screen.

Appellant respectfully notes that the seventh cited passage for elements 914 and 916 of Figure 9 (i.e., description in col. 21:20-25) teach the operation of the debugging engine of Leask. Specifically, although description of element 914 discloses that debugging can be halted if a debug tool condition is true, Leask does not teach that the debugging icon performs "one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 10.

Appellant notes that Leask discloses the use of user breakpoints, conditional breakpoints, and trigger breakpoints in column 13, lines 22 though column 14, line 10. However, Leask does not teach or suggest, in the cited passages or in the rest of the patent, the element of "based on the debugging results, performing one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 10.

Thus, Leask fails to teach the above features of claim 10, and so claim 10 is allowable.

**Claim 15**

In addition to the novel limitations of claims 1 and 13, Leask fails to teach or suggest "wherein said associating comprises:

receiving user input from a pointing device selecting the first data flow path in the first graphical program, wherein the first data flow path is configured to carry data of a first data type;

displaying a plurality of debugging graphical programs, wherein each of the plurality of debugging graphical programs is compatible with the first data type, and wherein the plurality debugging graphical programs comprises the debugging graphical program; and

receiving user input selecting the debugging graphical program from the plurality of debugging graphical programs", as recited in claim 15.

In asserting that Leask teaches these features, the Office Action cites col.19:5, col. 21:40-43 and 55-60, and col. 21:47-50.

> The debug engine is capable of suspending the application's execution at any given entry point in order to insert a desired debug tool at such an entry point.

(col. 19:3-6)

> Accordingly, each debug tool requested by a developer may have an <u>indicator displayed within the graphical representation of the application program</u> and may be communicated to the debug engine.

(col. 21:40-43)

> Thereafter, the developer may utilize various debug tools to analyze the application program's operation without interrupting the program's operation.

(col. 21: 55-60)

> Debugging an application program during runtime offers developers the ability to analyze the program's operation to detect and examine problems with the program without interrupting the program's operation. For example, suppose that a VRU application is running at a bank which allows the bank's customers to call the VRU and interact with the application by pressing touch tone keys on the customers' telephones or speaking verbal commands into the telephones. A customer may be able to press 1 to access the customer's savings account, press 2 to access the customer's checking account, press 3 to apply for a loan from the bank, and so on. Utilizing the graphical runtime debugger, a developer may view a graphical representation of the application program running on the bank's system.

(col. 21:42-52)

The second cited passage describes that a breakpoint may be indicated graphically in the graphical program of Leask. Specifically, the Summary of Leask describes that the graphical debugging environment "may display <u>indicators</u> illustrating where debug tools have been inserted within the application program." Therefore the cited indicator does not display data from the graphical program as a result of executing the debugging graphical program, but instead only indicates a location of the debugging icon in the graphical program of Leask.

Appellant notes that none of the cited passages disclose displaying debugging graphical program (i.e., debugging tools of Leask) that are compatible with the data type

of a first data flow path in the graphical program. Instead, the cited passages (i.e., col. 21:40-43 and 55-60) simply state that a developer can "utilize various debugging tools." This is very different from displaying debugging tools that are compatible with the selected data flow path of claim 15.

Appellant also notes that the last cited passage (col. 21:42-52) describes user operation of the graphical program, and does not describe user selection of the debugging icon. For example, the cited passage describes how a customer can press touch tone keys to access customer's bank account. Clearly, this passage has no bearing on the features and limitations of claim 15, as it does not apply to a user selecting a debugging program from a list of displayed debugging graphical programs that are compatible with the data type of a first data flow path in the graphical program.

Thus, Leask fails to teach all the features of claim 15, and so claim 15 is allowable.

**Claims 23, 25, and 26**

Claim 23 is separately patentable because the cited reference does not teach or suggest the limitations recited in this claim. Leask teaches using debugging tools for debugging a graphical representation of a program, where the debugging tools "may be represented graphically to indicate which tools are currently set in the program" (Leask col. 7, lines 26-29).

In asserting that Leask teaches the elements of claim 23, the Office Action cites col. 11:45-50, col. 7:7-20 & 10:1-5, elements 410, 412, 414, 416, and 418 of Figure 5, and col. 13:23-30, but fails to explain how any of the cited portions teach a debugging graphical program or creating a debugging graphical program using a graphical programming environment. The description for the cited elements 410, 412, 414, 416, and 418 of Figure 5, described in Leask col. 12, lines 18-36, reads:

> Turning to FIG. 5, an exemplary graphical debugging environment is shown. Call flow diagram 300, as discussed above with reference to FIG. 4, is shown. The graphical debugging environment is capable of displaying a graphical representation of an application program much like a graphical development environment. Additionally, various testing and debugging tools 402-418 that may be offered to a developer in such a graphical debugging environment are illustrated. The function of each testing and debugging

> tool will be discussed in greater detail hereafter. It is important to realize at this point that debugging tools may be utilized within the graphical debugging environment without requiring a programmer to interact with the underlying textual source code of an application program. That is, debugging tools may be utilized by a programmer to debug an application at the graphical representation level, rather than at the lower textual source code level. Accordingly, the graphical debugging environment allows a programmer to utilize graphical debugging tools to perform highlevel debugging.

Appellant notes that the cited elements 410, 412, 414, 416, and 418 of Leask are icons that represent executable functionality, and are not graphical programs that can be created by the user as recited in claim 23. Furthermore, the icons for the debugging tools of Leask do not contain interconnected graphical program nodes as required by claim 23 to be a graphical program. In other words, the debugging tools of Leask are not created using the same graphical programming environment as the graphical programs of Figures 3 or 4 of Leask. Figure 5 of Leask shows the debugging environment that debugs the graphical program of Figure 4 using debugging tools.

Claim 23 claims that the debugging graphical program is created using the graphical programming environment. Leask does not teach or suggest that any of the graphical representations, i.e., icons, of the debugging tools (i.e., the cited elements 410, 412, 414, 416, or 418 of Figure 5) were created using a graphical programming development environment. Specifically, the cited passage (col. 11, lines 45-50) does not show that the debugging icons 410, 412, 414, 416, or 418 of Leask were created using a graphical programming environment. Leask at column 11, lines 45-53 reads:

> Icon 330 represents the end of the call. Thus, call flow diagram 300 is an example of a graphical development environment in which a computer application is represented graphically. It should be understood that an application program need not be represented exactly as shown in FIGS. 3 and 4. Rather, an application may be represented in any manner, and the graphical debugger disclosed herein may be implemented to debug any manner of graphical representation.

The cited passage only teaches an exemplary graphical program of Leask, and it does not teach that the debugging icons of Leask were created using the graphical programming environment. Appellant notes that there are two graphical programs recited in claim 23, the first graphical program and the debugging graphical program. The cited passage only teaches that the debugging tools 410, 412, 414, 416, or 418 can debug a

graphical program, such as the one created using the graphical development environment shown in Figure 5 of Leask. In other words, although the debugging tools of Leask are represented as graphical icons (e.g., disclosed in col. 7, line 20-35), the icons are not themselves graphical programs. Leask simply does not teach or suggest that a user can create or modify a graphical program for a debugging icon. Thus, neither the icons that represent the debugging tools nor the debugging tools of Leask are executable graphical programs.

Appellant additionally notes that Figures 8 and 9 of Leask also are not graphical programs that contain programmable graphical program elements such as nodes and dataflow, as recited in claim 23, but instead are only diagrams that show the operation of the debugging engine of Leask.

Therefore Leask does not teach creating a debugging graphical program using a plurality of interconnected graphical program nodes or icons that graphically represent the functionality of the debugging graphical program. Thus, the graphical representation of the debugging tool of Leask is not equivalent to the debugging graphical program of claim 23.

In addition, claim 23 also contains novel elements of providing the data to a debugging graphical program, executing the debugging graphical program, where the debugging graphical program uses the data, and the debugging graphical program generating debugging results. Appellant notes that the Office Action rejection for claim 10 contains rejections for analogous elements of claim 23, and Appellant will address them here.

Appellant notes that the (col. 3:15-23) cited passage teaches generating a symbol table and mapping of a program image into a user process by the use of an image file. The (col. 3:15-23) cited passage does not teach or suggest "providing the data to the debugging graphical program." Appellant notes that the (col. 10:1-5) cited passage discloses distributed operation of the debugger of Leask. The (col. 10:1-5) cited passage does not teach or suggest "providing the data to the debugging graphical program." The (col. 3:15-23) cited passage and (col. 10:1-5) cited passage, taken singly or in

combination, do not teach or suggest "providing the data to the debugging graphical program."

Appellant notes that (col. 3:15-23) cited passage teaches generating a symbol table and mapping of a program image into a user process by the use of an image file. Appellant notes that the (col. 7:7-10) cited passage teaches displaying the graphical program to be debugged on the screen. The (col. 3:15-23) cited passage and (col. 7:7-10) cited passage, taken singly or in combination, do not teach or suggest "executing the debugging graphical program, wherein the debugging graphical program uses the data." Instead, the (col. 3:15-23) cited passage and (col. 7:7-10) cited passage teach that an object file that contains the graphical program may also contain a symbol file that holds the variables for the graphical program. This graphical program can then be displayed on the screen in graphical form. Although these passages also describe that a debugger can access the symbol table, it is clear from this passage that the symbol table that contains the data is generated for the object file for the graphical program prior to its execution and it is not generated for the debugging icon.

Thus, Leask fails to teach all the features of claim 23, and so claim 23 is allowable.

**Claim 24**

In addition to the novel limitations of claim 23, Leask fails to teach or suggest "

after the debugging graphical program generates debugging results, performing one or more of: halting execution of the first graphical program; entering a single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 24.

In asserting that Leask teaches this feature, the Office Action cites col. 7:7-10, elements 914 and 916 of Figure 9, and elements 414 and 416 of Figure 5, which read:

> In a preferred embodiment, a <u>graphical representation of an application program to be debugged</u> is displayed to a developer.

(col. 7:7-10)

> Step 414 is provided within the graphical debug environment. Step 414 may allow a developer to step through execution of the application program. For example, once a breakpoint fires to suspend execution of the application

> program, a developer may use Step 414 to step through the application program's execution. Accordingly, the application program may advance one icon (or one "block" of source code) each time that the developer activates Step 414. As a developer steps through each icon of a program the icon currently being executed may be highlighted or indicated in some other fashion to allow a developer to monitor the progress of the program's execution.
>
> Step Over 416 is also shown in FIG. 5. Step Over 416 may allow a developer to step from an icon (or "block" of source code) displayed on one level of the call flow to the next icon (or "block" of source code) displayed on the same level by stepping over all of the icons (or "blocks" of source code) on underlying levels of the call flow. For example, suppose a breakpoint is set at a graphical icon that contains several underlying icons. Once the breakpoint fires the application's execution will be suspended. A developer may use Step 414 to step through the execution of each graphical icon, including each underlying graphical icon. However, a developer may use Step Over 416 to advance the program's execution to the next graphical icon on the same level as the icon for which the breakpoint was set. Thus, Step Over 416 allows the application program to perform all of the underlying icons uninterrupted, and then suspend the application at the next icon at the same level as the icon for which the breakpoint was set. Again, the icon currently being executed may be highlighted or otherwise indicated to allow a developer to monitor the progress of the program's execution.

(col. 16:17-49)

> If the debug engine determines at block 912 that the debug tool's condition is true, the debug engine performs the debug tool's function at block 914. For example, for a breakpoint the debug engine may halt execution at block 914.

(col. 21:20-25)

Appellant notes that the first cited passage (col. 7:7-10) teaches displaying the graphical program to be debugged on the screen. The first cited passage does not teach or suggest "after the debugging graphical program generates debugging results, performing one or more of..." Instead, the first cited passage teaches that a graphical program can be displayed on the screen in graphical form. Although these passages also describe that a debugger can access the symbol table, it is clear from this passage that the symbol table that contains the data is generated for the object file for the graphical program prior to its execution and it is not generated for the debugging icon.

Appellant respectfully notes that the second cited passage for elements 914 and 916 of Figure 9 (i.e., description in col. 21:20-25) teach the operation of the debugging engine of Leask. Specifically, although description of element 914 discloses that debugging can be halted if a debug tool condition is true, Leask does not teach that the debugging icon performs "one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 24.

Appellant notes that Leask discloses the use of user breakpoints, conditional breakpoints, and trigger breakpoints in column 13, lines 22 though column 14, line 10. However, Leask does not teach or suggest, in the cited passages or in the rest of the patent, the element of "based on the debugging results, performing one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program", as recited in claim 24.

Thus, Leask fails to teach the above features of claim 24, and so claim 24 is allowable.

**Claims 35, 36, and 14**

Claim 35 is separately patentable because the cited reference does not teach or suggest the limitations recited in this claim. Leask teaches using debugging tools for debugging a graphical representation of a program, where the debugging tools "may be represented graphically to indicate which tools are currently set in the program" (Leask col. 7, lines 26-29).

In asserting that Leask teaches the elements of claim 35, the Office Action cites col. 11:45-50, col. 7:7-20 & 10:1-5, elements 410, 412, 414, 416, and 418 of Figure 5, col. 13:23-30, col. 10:5-10, col. 3:15-23, and elements 914 and 916 of Figure 9, but fails to explain how any of the cited portions teach a debugging graphical program or creating a debugging graphical program using a graphical programming environment.

The description for the cited elements 410, 412, 414, 416, and 418 of Figure 5, described in Leask col. 12, lines 18-36, reads:

> Turning to FIG. 5, an exemplary graphical debugging environment is shown. Call flow diagram 300, as discussed above with reference to FIG. 4, is shown. The graphical debugging environment is capable of displaying a graphical representation of an application program much like a graphical development environment. Additionally, various testing and debugging tools 402-418 that may be offered to a developer in such a graphical debugging environment are illustrated. The function of each testing and debugging tool will be discussed in greater detail hereafter. It is important to realize at this point that debugging tools may be utilized within the graphical debugging environment without requiring a programmer to interact with the underlying textual source code of an application program. That is, debugging tools may be utilized by a programmer to debug an application at the graphical representation level, rather than at the lower textual source code level. Accordingly, the graphical debugging environment allows a programmer to utilize graphical debugging tools to perform highlevel debugging.

Appellant notes that the cited elements 410, 412, 414, 416, and 418 of Leask are icons that represent executable functionality, and are not graphical programs that can be created by the user as recited in claim 35. Furthermore, the icons for the debugging tools of Leask do not contain interconnected graphical program nodes as required by claim 35 to be a graphical program. In other words, the debugging tools of Leask are not created using the same graphical programming environment as the graphical programs of Figures

3 or 4 of Leask. Figure 5 of Leask shows the debugging environment that debugs the graphical program of Figure 4 using debugging tools.

Claim 35 claims that the second graphical program is created using the graphical programming environment. Leask does not teach or suggest that any of the graphical representations, i.e., icons, of the debugging tools (i.e., the cited elements 410, 412, 414, 416, or 418 of Figure 5) were created using a graphical programming development environment. Specifically, the cited passage (col. 11, lines 45-50) does not show that the debugging icons 410, 412, 414, 416, or 418 of Leask were created using a graphical programming environment. Leask at column 11, lines 45-53 reads:

> Icon 330 represents the end of the call. Thus, call flow diagram 300 is an example of a graphical development environment in which a computer application is represented graphically. It should be understood that an application program need not be represented exactly as shown in FIGS. 3 and 4. Rather, an application may be represented in any manner, and the graphical debugger disclosed herein may be implemented to debug any manner of graphical representation.

The cited passage only teaches an exemplary graphical program of Leask, and it does not teach that the debugging icons of Leask were created using the graphical programming environment. Appellant notes that there are two graphical programs recited in claim 35, the first graphical program and the second graphical program. The cited passage only teaches that the debugging tools 410, 412, 414, 416, or 418 can debug a graphical program, such as the one created using the graphical development environment shown in Figure 5 of Leask. In other words, although the debugging tools of Leask are represented as graphical icons (e.g., disclosed in col. 7, line 20-35), the icons are not themselves graphical programs. Leask simply does not teach or suggest that a user can create or modify a graphical program for a debugging icon. Thus, neither the icons that represent the debugging tools nor the debugging tools of Leask are executable graphical programs.

Appellant additionally notes that Figures 8 and 9 of Leask also are not graphical programs that contain programmable graphical program elements such as nodes and dataflow, as recited in claim 35, but instead are only diagrams that show the operation of the debugging engine of Leask.

Therefore Leask does not teach creating a second graphical program using a plurality of interconnected graphical program nodes or icons that graphically represent the functionality of the second graphical program. Thus, the graphical representation of the debugging tool of Leask is not equivalent to the second graphical program of claim 35.

Claim 35 also contains novel elements of "a data structure which is operable to store information associating the second graphical program with a location in the first graphical program, wherein the functionality of the first graphical program is not modified by the second graphical program." Appellant notes that the Office Action rejection for claim 35 also cites rejections for claim 10. Appellant respectfully notes that claim 35 does not contain the elements of claim 10. Appellant respectfully notes that the dependent claim 36 is analogous to dependent claim 10. Therefore analysis of the rejection for claim 10 is not necessary with regard to claim 35.

Appellant notes that the Office Action rejection for claim 14, which also recites a data structure used in "associating," cites Leask at column 10:1-10. The cited passage of Leask reads:

> An application program to be debugged may be running either on the local system $100_{LOCAL}$ or on a remote system, such as $100_{REMOTE}$ shown in FIG. 1. Accordingly, the <u>graphical debugger program running on local computer $100_{LOCAL}$ may be utilized to debug locally stored programs or programs stored at remote locations</u> (e.g., $100_{REMOTE}$) via network 108.
> Turning to FIG. 2, an example of how a prior art textual debugger may be utilized is shown.

(col. 10:1-10)

Appellant notes that the cited passage (col. 10:1-10) discloses distributed operation of the debugger of Leask. The first cited passage does not teach or suggest "a data structure which is operable to store information associating the second graphical program with a location in the first graphical program." Appellant respectfully notes that the cited passage does not teach a "data structure" at all.

Thus, Leask fails to teach all the features of claim 35, and so claim 35 is allowable.

**Second Ground of Rejection**

Claim 8 stands rejected under 35 U.S.C. 103(a) as being unpatentable over Leask in view of McKee. Appellant respectfully traverses this rejection for the following reasons.

**Claim 8**

The Office Action admits that Leask fails to teach or suggest "wherein said statistics comprise one or more of:

data generated by the debugging graphical program;

data generated by a plurality of executions of the debugging graphical program during a corresponding plurality of executions of the first graphical program, wherein said data generated by the plurality of executions of the debugging graphical program includes differences in execution times between the plurality of executions of the debugging graphical program, wherein said differences in execution times are useable in optimizing performance the first graphical program," but asserts that McKee remedies this admitted deficiency of Leask, citing McKee at col. 3:42-47.

McKee at col. 3:41-47 reads:

> The dynamic trace-driven code optimizer gathers runtime data about the execution paths the program is following as the program is being run. It then analyzes the code being executed for non-optimal instruction streams and modifies the code in real-time in order to generate optimized object code that is capable of enhanced performance for the given data being run with the program.

Appellant respectfully submits that McKee does not disclose this feature of claim 8, but rather, the cited passage is directed to dynamic optimization of the object code being executed. In fact, McKee teaches away from using execution statistics and accessing source code. Instead, McKee teaches to dynamically analyze object code of a program during its execution, whereas claim 8 claims using data that is generated by executions of the debugging graphical program that correspond to executions of the first graphical program. Furthermore, McKee does not disclose anywhere using differences in execution times between a plurality of executions of a program.

Thus, Leask and McKee fail to teach all the features and limitations of claim 8.

Moreover, Appellant submits that the Office Action has not provided a proper motivation to combine Leask and McKee. The only motivation to combine suggested by the Office Action is that it would have been obvious to one of ordinary art. Appellant respectfully submits that the Office Action has simply stated a benefit of Appellant's invention based on the subject matter of claim 8, and has not provided any teaching or suggestion to combine from the references themselves, which is improper. Thus, the attempted combination of Leask and McKee is not available for making a prima facie case of obviousness.

Finally, Appellant notes that even were Leask and McKee properly combinable, which Appellant argues they are not, the resulting combination would still not produce Appellant's invention as claimed, as argued above, and so Appellant respectfully submits that claim 8 is allowable for at least the reasons provided.

**Third Ground of Rejection**

Claim 16 stands rejected under 35 U.S.C. 103(a) as being unpatentable over Leask in view of Kodosky. Appellant respectfully traverses this rejection for the following reasons.

**Claim 16**

The Office Action admits that Leask fails to teach or suggest "wherein said associating comprises:

receiving user input selecting the first data flow path in the first graphical program, wherein the first data flow path is configured to carry data of a first data type;

determining the first data type of the first data flow path;

displaying a plurality of debugging graphical programs appropriate for the first data type of the first data flow path; and

receiving user input selecting the debugging graphical program from the plurality of debugging graphical programs appropriate for the first data type of the first data flow path," but asserts that Kodosky remedies this admitted deficiency of Leask, citing Kodosky at paragraph [032].

Kodosky at paragraph [032] reads:

> Where the program icon corresponds to a graphical program, this association may cause the block diagram corresponding to this program icon to automatically be displayed. In this instance, the user may then further graphically navigate, e.g., move or drag, the device icon within the block diagram that has been displayed and drop or place the device icon at a respective location in the graphical program. This may cause the device icon to be copied or inserted into the displayed block diagram at the selected location. Alternatively, a graphical program node may be displayed in the diagram which is associated with or operable to access or invoke functionality of the device, and the user may position this node at a desired location in the block diagram. The user may select a flow path, such as a data flow wire, in which to position or "drop" the device icon (or node). The device icon may then be inserted on to or in the execution or data path of the selected wire in the block diagram of the graphical program and configured to execute. The user may also manually connect or wire the device icon with other nodes in the diagram.

Appellant respectfully submits that Kodosky does not disclose this feature of claim 16, but rather, the cited passage is directed to using a configuration diagram and specifically to displaying block diagrams that are associated with a program icon on a configuration diagram. Nowhere in the cited passage does Kodosky teach or suggest "displaying a plurality of debugging graphical programs appropriate for the first data type of the first data flow path, and receiving user input selecting the debugging graphical program from the plurality of debugging graphical programs appropriate for the first data type of the first data flow path." The cited Kodosky patent does not disclose displaying or using debugging graphical programs, except for Remote Debugging described in paragraphs [0446] through [0448], which does not use debugging graphical programs as claimed in claim 16.

Thus, Leask and Kodosky fail to teach all the features and limitations of claim 16.

Moreover, Appellant submits that the Office Action has not provided a proper motivation to combine Leask and Kodosky. The only motivation to combine suggested by the Office Action is that it would have been obvious to one of ordinary art. Appellant respectfully submits that the Office Action has simply stated a benefit of Appellant's invention based on the subject matter of claim 16, and has not provided any teaching or suggestion to combine from the references themselves, which is improper. Thus, the attempted combination of Leask and Kodosky is not available for making a prima facie case of obviousness.

Appellant also notes that even were Leask and Kodosky properly combinable, which Appellant argues they are not, the resulting combination would still not produce Appellant's invention as claimed, as argued above, and so Appellant respectfully submits that claim 16 is allowable for at least the reasons provided.

Finally, Appellant notes that the Kodosky reference is not available as prior art since both the current application and the Kodosky reference have the same priority date of August 14, 2001.

For the foregoing reasons, it is submitted that the Office Action's rejection of claims 1-11 and 13-40 was erroneous, and reversal of the decision is respectfully requested.

If any extensions of time (under 37 C.F.R. § 1.136) are necessary to prevent the above-referenced application(s) from becoming abandoned, Applicant(s) hereby petition for such extensions. The Commissioner is hereby authorized to charge any fees which may be required or credit any overpayment to Meyertons, Hood, Kivlin, Kowert & Goetzel P.C., Deposit Account No. 50-1505/5150-59901/JCH.

Respectfully submitted,

/Jeffrey C. Hood/

Jeffrey C. Hood, Reg. #35198
ATTORNEY FOR APPLICANT(S)

Meyertons Hood Kivlin Kowert & Goetzel, P.C.
P.O. Box 398
Austin, TX 78767-0398
Phone: (512) 853-8800

Date: March 29, 2007 JCH/MRW

## VIII. CLAIMS APPENDIX

The claims on appeal are as follows:

1. A computer-implemented method for creating a graphical program, the method comprising:

creating a first graphical program using a graphical programming development environment, wherein said creating comprises interconnecting at least two of a first plurality of graphical program nodes or icons, wherein the first graphical program comprises the first plurality of interconnected graphical program nodes or icons which graphically represents functionality of the first graphical program, and wherein the first graphical program is executable by a computer system to perform the functionality;

storing the first graphical program in a memory; and

associating a debugging graphical program at a debugging location in the first graphical program, wherein said associating does not modify the functionality of the first graphical program, wherein the debugging graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the debugging graphical program, wherein the debugging graphical program was created using the graphical programming development environment;

wherein the debugging graphical program is executable during execution of the first graphical program to aid in debugging at least a portion of the first graphical program.

2. The computer-implemented method of claim 1, wherein said associating does not require a re-compilation of the first graphical program.

3. The computer-implemented method of claim 1, further comprising:

executing the first graphical program up to the debugging location;

executing the debugging graphical program after executing the first graphical program up to the debugging location; and

the debugging graphical program generating debugging results, wherein the debugging results are useful in analyzing at least a portion of the first graphical program.

4. The computer-implemented method of claim 3, further comprising:

completing execution of the first graphical program based on the debugging results of said executing the debugging graphical program.

5. The computer-implemented method of claim 3,

wherein said executing the debugging graphical program includes displaying the debugging results of the debugging graphical program.

6. The computer-implemented method of claim 3, wherein said executing the debugging graphical program comprises:

receiving data from the first graphical program; and

performing one or more of:

displaying the data from the first graphical program; and/or

logging the data from the first graphical program to a file.

7. The computer-implemented method of claim 3, wherein said executing the debugging graphical program comprises:

receiving data from the first graphical program;

generating statistics based on the received data; and

displaying the statistics.

8. The computer-implemented method of claim 7, wherein said statistics comprise one or more of:

data generated by the debugging graphical program;

data generated by a plurality of executions of the debugging graphical program during a corresponding plurality of executions of the first graphical program, wherein said data generated by the plurality of executions of the debugging graphical program includes differences in execution times between the plurality of executions of the

debugging graphical program, wherein said differences in execution times are useable in optimizing performance the first graphical program.

9. The computer-implemented method of claim 3,

wherein said completing execution of the first graphical program is performed in a single stepping mode based on the debugging results of said executing the debugging graphical program.

10. The computer-implemented method of claim 1, further comprising:

executing the first graphical program up to the debugging location, wherein the first graphical program generates data at the debugging location;

providing the data to the debugging graphical program;

executing the debugging graphical program, wherein the debugging graphical program uses the data;

the debugging graphical program generating debugging results;

based on the debugging results, performing one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program.

11. The computer-implemented method of claim 10, wherein the first graphical program executes up to the debugging location where the debugging graphical program is associated, and waits for user input.

12. (Cancelled)

13. The computer-implemented method of claim 1,

wherein the first graphical program comprises a plurality of data flow paths;

wherein said associating the debugging graphical program at the location in the first graphical program comprises associating the debugging graphical program at a first data flow path in the first graphical program.

14. The computer-implemented method of claim 13, wherein said associating comprises:

storing information in at least one data structure, wherein the information comprises information regarding the first graphical program, the debugging graphical program, and the location where the debugging graphical program is attached along the first data flow path of the first graphical program.

15. The computer-implemented method of claim 13, wherein said associating comprises:

receiving user input from a pointing device selecting the first data flow path in the first graphical program, wherein the first data flow path is configured to carry data of a first data type;

displaying a plurality of debugging graphical programs, wherein each of the plurality of debugging graphical programs is compatible with the first data type, and wherein the plurality debugging graphical programs comprises the debugging graphical program; and

receiving user input selecting the debugging graphical program from the plurality of debugging graphical programs.

16. The computer-implemented method of claim 13, wherein said associating comprises:

receiving user input selecting the first data flow path in the first graphical program, wherein the first data flow path is configured to carry data of a first data type;

determining the first data type of the first data flow path;

displaying a plurality of debugging graphical programs appropriate for the first data type of the first data flow path; and

receiving user input selecting the debugging graphical program from the plurality of debugging graphical programs appropriate for the first data type of the first data flow path.

17. The computer-implemented method of claim 1, wherein said associating the debugging graphical program at the debugging location in the first graphical program comprises associating the debugging graphical program at a node or icon in the first graphical program.

18. The computer-implemented method of claim 1, further comprising:

disassociating the debugging graphical program from the first graphical program, wherein said disassociating does not modify the first graphical program and/or does not require a re-compilation of the first graphical program.

19. The computer-implemented method of claim 1,

wherein the first graphical program is located on a first computer system;

wherein the debugging graphical program is located on a second computer system, wherein the second computer system is coupled to the first computer system over a network.

20. The computer-implemented method of claim 19, the method further comprising:

executing the first graphical program on the first computer system up to the debugging location;

executing the debugging graphical program on the second computer system, wherein the debugging graphical program is executed after executing the first graphical program on the first computer system up to the debugging location;

the debugging graphical program generating debugging results on the second computer system; and

providing the debugging results from the second computer system to the first computer system.

21. The computer-implemented method of claim 1,

wherein the first graphical program is located on a first computer system, wherein the first computer system is a target computer system coupled to or comprised in a second computer system;

wherein the debugging graphical program is located on and executed on the first computer system.

22. The computer-implemented method of claim 1,

wherein the first graphical program is located on a first computer system, wherein the first computer system is a target computer system coupled to or comprised in a second computer system;

wherein the debugging graphical program is located on and executed on the second computer system.

23. A computer-implemented method for executing a first graphical program, the method comprising:

executing the first graphical program up to a debugging location, wherein the first graphical program comprises a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and wherein the first graphical program is executable by a computer system to perform the functionality, wherein the first graphical program generates data at the debugging location, wherein the first graphical program was created using a graphical programming development environment;

providing the data to a debugging graphical program, wherein the debugging graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the debugging graphical program, wherein the debugging graphical program was created using the graphical programming development environment;

executing the debugging graphical program, wherein the debugging graphical program uses the data;

the debugging graphical program generating debugging results;

wherein use of the debugging graphical program does not require modification or re-compilation of the first graphical program.

24. The computer-implemented method of claim 23,

after the debugging graphical program generates debugging results, performing one or more of: halting execution of the first graphical program; entering a single stepping mode in the first graphical program; and/or completing execution of the first graphical program.

25. The computer-implemented method of claim 23, further comprising:

associating the debugging graphical program at the debugging location in the first graphical program;

wherein said associating does not require modification or recompilation of the first graphical program.

26. The computer-implemented method of claim 23,

wherein the first graphical program is located on a first computer system;

wherein the debugging graphical program is located on a second computer system, wherein the second computer system is coupled to the first computer system over a network;

wherein the first graphical program executes on the first computer system up to the debugging location;

wherein the debugging graphical program executes on the second computer system, wherein the debugging graphical program is executed after executing the first graphical program on the first computer system up to the debugging location; and

wherein the debugging graphical program generates debugging results on the second computer system.

27. A computer-implemented method for analyzing a first graphical program, the method comprising:

storing the first graphical program in a memory of a computer system, wherein the first graphical program comprises a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and wherein the first graphical program is executable by the computer system to perform the functionality, wherein the first graphical program was created using a graphical programming development environment;

associating a second graphical program at a location in the first graphical program, wherein said associating does not modify the functionality of the first graphical program, wherein the second graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program, wherein the second graphical program was created using the graphical programming development environment;

wherein the second graphical program is executable during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program.

28. The computer-implemented method of claim 27, wherein said associating does not require a re-compilation of the first graphical program.

29. The computer-implemented method of claim 27, further comprising:

executing the first graphical program up to the location;

executing the second graphical program after executing the first graphical program up to the location; and

the second graphical program generating results, wherein the results are useful in analyzing at least a portion of the first graphical program.

30. he computer-implemented method of claim 27, further comprising:

executing the first graphical program up to the location, wherein the first graphical program generates data at the location;

providing the data to the second graphical program;

executing the second graphical program, wherein the second graphical program uses the data;

the second graphical program generating results;

based on the results, performing one or more of: halting execution of the first graphical program; entering a single stepping mode in the first graphical program; and/or completing execution of the first graphical program.

31. A memory medium comprising program instructions for analyzing a first graphical program, wherein the program instructions are executable to implement:

storing the first graphical program in a memory of a computer system, wherein the first graphical program comprises a first plurality of interconnected graphical program nodes or icons that graphically represents functionality of the first graphical program, and wherein the first graphical program is executable by the computer system to perform the functionality, wherein the first graphical program was created using a graphical programming development environment;

associating a second graphical program at a location in the first graphical program, wherein said associating does not modify the functionality of the first graphical program, wherein the second graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program, wherein the second graphical program was created using the graphical programming development environment;

wherein the second graphical program is executable during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program.

32. The memory medium of claim 31, wherein said associating does not require a re-compilation of the first graphical program.

33. The memory medium of claim 31, wherein the program instructions are further executable to implement:

executing the first graphical program up to the location;

executing the second graphical program after executing the first graphical program up to the location; and

the second graphical program generating results, wherein the results are useful in analyzing at least a portion of the first graphical program.

34. The memory medium of claim 31, wherein the program instructions are further executable to implement:

executing the first graphical program up to the location, wherein the first graphical program generates data at the location;

providing the data to the second graphical program;

executing the second graphical program, wherein the second graphical program uses the data;

the second graphical program generating results;

based on the results, performing one or more of: halting execution of the first graphical program; entering a single stepping mode in the first graphical program; and/or completing execution of the first graphical program.

35. A memory medium comprising:

a first graphical program, wherein the first graphical program comprises a first plurality of interconnected nodes which visually indicate functionality of the first graphical program, and wherein the first graphical program is executable by a computer system to perform the functionality, wherein the first graphical program was created using a graphical programming development environment;

a second graphical program, wherein the second graphical program comprises a second plurality of interconnected graphical program nodes or icons that graphically represents functionality of the second graphical program, wherein the second graphical program was created using the graphical programming development environment;

a data structure which is operable to store information associating the second graphical program with a location in the first graphical program, wherein the

functionality of the first graphical program is not modified by the second graphical program;

wherein the second graphical program is executable during execution of the first graphical program to aid in analyzing at least a portion of the first graphical program.

36. The memory medium of claim 35, further comprising program instructions which are executable to:

execute the first graphical program up to the location, wherein the first graphical program generates data at the location;

provide the data to the second graphical program;

execute the second graphical program, wherein the second graphical program uses the data and generates results;

based on the results, perform one or more of: halting execution of the first graphical program; entering single stepping mode in the first graphical program; and/or completing execution of the first graphical program.

37. The computer-implemented method of claim 1, wherein said associating the debugging graphical program at the debugging location in the first graphical program comprises:

including the debugging graphical program at the debugging location in the first graphical program.

38. The computer-implemented method of claim 23, further comprising:

including the debugging graphical program at a debugging location in the first graphical, wherein said including does not modify the functionality of the first graphical program.

39. The computer-implemented method of claim 27, wherein said associating the second graphical program at the location in the first graphical program comprises:

including the second graphical program at the location in the first graphical program.

40. The memory medium of claim 31, wherein, in said associating the second graphical program at the location in the first graphical program comprises, the program instructions are further executable to implement:

including the second graphical program at the location in the first graphical program.

## IX. EVIDENCE APPENDIX

No evidence submitted under 37 CFR §§ 1.130, 1.131 or 1.132 or otherwise entered by the Examiner is relied upon in this appeal.

## X. RELATED PROCEEDINGS APPENDIX

There are no related proceedings.